# Automated Analysis of Virtual Prototypes at Electronic System Level

Mehran Goli, Muhammad Hassan, Daniel Große, Rolf Drechsler
Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
Institute of Computer Science, University of Bremen, 28359 Bremen, Germany

{mehran.goli,muhammad.hassan}@dfki.de,{grosse,drechsle}@cs.uni-bremen.de

## ABSTRACT

The exponential increase in functionality of *System-on-Chips* (SoCs) and reduced *Time-to-Market* (TTM) requirements have significantly altered the typical design and verification flow. *Virtual Prototyping* (VP) at the *Electronic System Level* (ESL) using SystemC and its *Transaction Level Modeling* (TLM) framework is an industry-accepted solution. VP design exploration, review, debugging, and integration of ever changing functional requirements can be made faster with the help of design understanding and visualization methods. Hence, in this paper, we propose a fully automated structural, and behavioral analysis approach for visualization of ESL VPs including TLM-2.0 VPs. At the heart of the analysis is a hybrid approach which uses static and dynamic methods to extract structural and behavioral information of the VP. Afterwards, the extracted information is translated into structural and graphical representations such as UML diagrams (specifying TLM-2.0 transactions' protocols), and XML format (describing designs' structure). Experimental results including a real-world VP shows the effectiveness of our approach.

**ACM Reference Format:**
Mehran Goli, Muhammad Hassan, Daniel Große, Rolf Drechsler. 2019. Automated Analysis of Virtual Prototypes at Electronic System Level. In *Great Lakes Symposium on VLSI 2019 (GLSVLSI '19), May 9–11, 2019, Tysons Corner, VA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3299874.3318024

## 1 INTRODUCTION

The increasing functionality of *System-on-Chips* (SoCs) and reduced *Time-to-Market* (TTM) constraints have significantly altered the design and verification flow to meet the high market demand. This has led to the rapidly-growing adoption of *Virtual Prototypes* (VPs) at *Electronic System Level* (ESL) abstraction in the last decade. Essentially, a VP is an abstract, and executable software model written in SystemC [1] using its *Transaction Level Modeling* (TLM) [2] framework. The much earlier availability as well as the significantly faster simulation speed in comparison to the *Register Transfer Level* (RTL) models are among the main benefits of VPs. These enable hardware/software co-design and verification very early in the development flow. As a consequence, VPs are heavily used for early architecture exploration for the next generation of SoCs. This allows the systems architects to test new features and validate the capabilities of the SoC, i.e. to decide where to reuse existing *Intellectual Properties* (IPs) and where to create new hardware blocks. However, this decision making requires design understanding of the SoC, which includes both, structural as well as behavioral insight. For a complex SoC with hundreds of IPs it is very hard and time consuming to understand the interfacing, and communication patterns of the VP. Therefore, the first logical step is to consult the available documentation, but, there may be scenarios, e.g., third party IPs, or legacy models, where the documentation is poorly written, or not available at all [14]. Thus, the development team members spend significant time to reverse engineer the design, hence, increasing time-to-market. This has raised the need for design understanding and visualization methods which enhance the design experience, debugging facilities, and integrated development environments. However, one major challenge specific to behavioral understanding of the TLM-2.0 models is the communication interface. The VP models communicate through thousands of TLM-2.0 transactions which are difficult to follow, redundant, and unnecessary from the perspective of design understanding. A concise visualization of the underlying VP transactions flow can accelerate the revisions and additions, while helping significantly in design understanding. While several approaches exist; static [5, 9, 10, 14], and hybrid [4, 6–8, 11, 13, 16], for information extraction, analysis, and visualization of SystemC VPs, they have the following drawbacks: 1) no support for TLM-2.0 constructs [4, 6, 7, 13], 2) lack of precise behavior extraction (the extracted information mostly describes the designs' structure and do not trace their transactions or variable's state) [4, 6, 13, 14, 16], 3) low degree of automation [11], and 4) scalability [7, 8].

In this paper, we propose an automated hybrid data extraction and analysis approach for visualization of ESL VPs. Our approach extracts the structural, and behavioral information of the VP using static and dynamic methods. The analysis reduces the amount of redundant transactions generated by VPs to a small amount of unique transactions. This gives the designer a concise view of the complete VP model. Our approach consists of two phases, and focuses on TLM-2.0 constructs in particular. In phase one, static analysis is performed on the given VP to extract the structural information, and VP instrumentation is done for dynamic run-time behavioral information extraction. In phase two, the instrumented VP is executed, and run-time information is analyzed. For design understanding intention, the retrieved structural information is presented in an XML format. The extracted behavioral information is translated into a set of *Unified Modeling Language* (UML) diagrams. The UML diagrams defined in this paper reflect the behavioral information in a big scale view to empower designers to easily trace both transactions' data and flow at the same time. Moreover, the retrieved intermediate structured logs allow designers to use them for tasks of debugging, validation and verification off-the-shelf or with minimum translation effort into their desired forms. The approach is applied to several case studies including a real-world VP to show its precision, scalability, and advantages.

## 2 MOTIVATION

Consider the third Party *AT_BUS* VP (inspired from [3]) shown in Fig. 1 where the documentation is not available (or poorly written).


*GLSVLSI '19, May 9–11, 2019, Tysons Corner, VA, USA*

ACM ISBN 978-1-4503-6252-8/19/05…$15.00
https://doi.org/10.1145/3299874.3318024

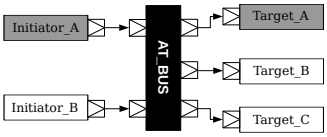


**Figure 1: The architecture of the *AT_BUS* VP.**

The VP includes six modules which differentiate on the basis of underlying base protocol transactions: two initiators (*Initiator_A*, *Initiator_B*), one interconnect (*AT_BUS*), and three targets (*Target_A*, *Target_B*, *Target_C*). *Initiator_A* communicates with target modules through *AT_BUS* by generating four types of *Approximately-timed* (AT) transactions. Two types $T_1$ and $T_2$ to access *Target_A* which is a memory (each type for different memory address ranges), and types $T_3$ and $T_4$ to access *Target_B* and *Target_C*, respectively. *Initiator_B* only generates transactions of type $T_4$ to communicate with all target modules. For example, consider the communication between *Initiator_A* and *Target_A* (the gray components in Fig. 1). The *Initiator_A* module generates transaction types $T_1$ and $T_2$ to access memory address range ($0x00$ to $0x0A$) and ($0x0B$ to $0xFF$) of the *Target_A* module based on the function calls and timing phases described in Table 1, respectively. Now consider a possible scenario that can happen during the design process.

**Scenario:** designers decide to reuse or revise some modules of the VP. For example, they want to modify the AT base protocol transaction type $T_1$ of the *Initiator_A* module and change it to $T_4$ including less transition phases to gain performance. This modification also needs to be applied to the *AT_BUS* and *Target_A* to properly build a communication path between the initiator and target modules through the interconnect. Before any changes can be applied to the VP, it needs to be properly understood. However, lacking a (good) documentation makes the understanding process very complex.

The first and common solution to handle the aforementioned scenarios is to use reverse engineering such as analyzing and instrumenting the existing source code and SystemC library (if required). This usually results in incomplete logs, work overhead, and waste of time in case of complex designs. In order to help designers in understanding the complexity of a given SystemC VP (especially for the third party or legacy designs), a SystemC analyzer tool is required that can properly extract and analyze both structure and behavior of the VP. The former refers to the modules' names, types (initiator, interconnect or target) and, instances and, binding information of modules' sockets. The latter refers to the three essential elements: transactions' *flow*, *data*, and *type*. The transaction's flow represents the order of TLM modules taking part in the transaction's lifetime (i.e. the period of time between transaction construction and destruction). For example the transaction's flow for both transaction types $T_1$ and $T_2$ generated by *Initiator_A* to access data in *Target_A* is based on the sequence order (1)→(2)→(3)→(2)→(1)→(2)→(3)→(2)→(1) where: (1) *Initiator_A*, (2) *AT_BUS* and, (3) *Target_A*. The transaction data denotes the transaction attributes such as data, address, length etc. The transaction's type refers to the transaction's timing model (Loosely-timed (LT) or AT) and in case of the AT model, it also specifies which type of the base protocol transactions is used. However, this element has not been considered by the existing SystemC analysis approaches [11, 14] as they only extract the transactions' flow (but not their type). The two transaction types $T_1$ and $T_2$ have the same communication pattern (flow) but their timing phases are different, these methods fail to distinguish between them. Therefore, the proposed method must be able to extract all the aforementioned elements, distinguish the transactions' unique type and flow and finally present them in a proper graphical format (e.g. UML diagrams), thus, designers can quickly grasp the VP behavior.

**Table 1: Transactions timing models of the *AT-BUS* VP**

| Type | Communication Interface Call | Return Status | Phase Transition |
|---|---|---|---|
| $T_1$ | nb_transport_fw/nb_transport_bw | TA→TU | BRQ→BRP→ERP |
| $T_2$ | nb_transport_fw/nb_transport_bw | TA→TC | BRQ→BRP |
| $T_3$ | nb_transport_fw | TU→TC | BRQ→BRP→ERP |
| $T_4$ | nb_transport_fw | TC | BRQ |

**TC:** TLM_COMPLETED **TA:** TLM_ACCEPTED **TU:** TLM_UPDATED **BRQ:** BEGIN_REQUEST
**BRP:** BEGIN_RESPONSE **ERP:** END_RESPONSE

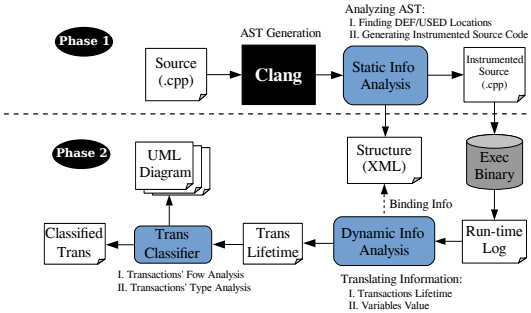


**Figure 2: The architecture of the proposed approach.**

# 3 METHODOLOGY

## 3.1 Overall Workflow

Fig. 2 provides an overview of the proposed approach including:

(1) analyzing the AST of a given VP to obtain two goals:
  (a) extracting the static information of the model which is required to describe design's structure and
  (b) using this information as the foundation for retrieving run-time information (i.e. behavior) by generating an instrumented version of the source code.
(2) retrieving the run-time information of the design by executing the instrumented binary model and translating it into structural formats.

## 3.2 Phase 1: Extracting Static Information

The extraction process is performed by visiting relevant nodes in the AST. As the top level entities of a VP are modules and global functions, the first entry point of extracting design's structure is to find the node including the information of the aforementioned entities. The information of modules' sockets (or signal ports), transaction (or variable) and member functions is also retrieved in the same way by visiting the corresponding nodes in the AST. This information is used to retrieve part of the design's structure (which is not available statically) and behavior (i.e. tracing transactions or variables values) at run-time. To do this, an instrumented version of the existing source code is automatically generated from the AST including *retrieving* statements. The statements are defined based on hierarchical structure where for tracing e.g. a transaction, the value of transaction's attributes and its related parameters such as timing annotation, phase (e.g. BEGIN_REQ) and functions' return status (e.g. TLM_COMPLETED) – for the AT model – are retrieved during execution. Moreover, the simulation time is extracted to notify the exact time of transaction or variable value changes. To trace a transaction after any possible change, we define two locations DEF and USED. The DEF location refers to the line of code where the transaction is defined (e.g. as a function arguments or a local variables within the function's body). In case of TLM-2.0 designs, the USED location refers to function calls (e.g. transport interfaces *b_transport* or *nb_transport*) where the transaction object is used as an input argument. Thus, the *retrieving* statements are inserted to the source code after the aforementioned locations.

```
1  struct Initiator_A: sc_module{
2  tlm_utils::simple_initiator_socket<Initiator_A, 32> init_socket;
3  ...
4  void thread_process(){
5  tlm::tlm_generic_payload* trans;
6  tlm::tlm_phase phase;
7  sc_time delay;
8  ...
9  status = init_socket->nb_transport_fw(*trans, phase, delay);
10 Fout<<"Initiator_A::thread_process::trans.ID="<<trans<<"DATA="<<trans->
       get_data_ptr()<<"CMD="<<trans->get_command()<<"ADR="<<trans->
       get_address()<<"RSP="<<trans->get_response_status()<<"DL="<<trans->
       get_data_length()<<"delay="<<delay<<"phase="<<phase<<"
       instance_name_module="<<this->name()<<"ST="<<sc_time_stamp()<<endl;
11 ...}
```

**Figure 3: A part of the instrumented source code of module *Initiator_A* of the *AT_BUS* VP.**

For example, consider a part of the source code related to the module `Initiator_A` of the *AT_BUS* VP (Fig. 3). Line 10 is not initially available. Assume that we want to trace all transactions generated by the `thread_process` function of the `Initiator_A` module. To do this, the VP's AST is analyzed by *Static Info Analysis* module (Fig. 2, Phase 1) to find DEF and USED locations in the source code. For example, consider a USED location (Line 9, in Fig. 3) where transaction `trans` is used as a function argument of the `nb_transport_fw` interface. To properly trace the transactions, all information related to the transactions' flow, data, and type must be extracted. This includes: 1) the module name (`Initiator_A`) and the parent function (`thread_process`) to which this transaction belongs and the transactions' reference address, 2) all attributes of the transactions which are data, address, response status, data length, and 3) transactions'related parameters which are the `phase` and `delay` arguments of the `nb_transport_fw` interface and its return status stored in the `status` variable. From the extracted information, the *retrieving* statement `Fout` (Line 10, Fig. 3) is automatically generated and inserted after the USED location in the new source code. The instructions `this->name()` and `sc_time_stamp()` are also added to the *retrieving* statement to identify that the transaction `trans` belongs to which instance of the `Initiator_A` module and the simulation time when the transaction is sent through the initiator socket `init_socket`, respectively.

## 3.3 Phase 2: Extracting Run-time Information

After generation of the instrumented source code, it is then automatically compiled with a standard C++ compiler (e.g. GCC or Clang) and executed to log the run-time information. The extracted information is translated into two formats XML (to reflect designs' structure) and UML (TLM-2.0 designs' behavior).

*3.3.1 Structure Presentation.* Major part of the designs' structure is extracted during the static analysis in the first phase. This information includes:

- the root name and type (for TLM modules can be initiator, interconnect or target which is identified by analyzing modules' sockets type) of each module,
- the name and type of each function,
- the variables (or signals) of each module and
- local variables of each function

The structural data that cannot be extracted during the static analysis is retrieved at run-time. This information (e.g. the instance name of modules and binding information of modules' ports and sockets) is extracted by *Dynamic Info Analysis* module from the *Run-time Log* and bounded to the static data. The final result is presented as an XML formatted file.

*3.3.2 Behavior Presentation.* Since the extracted information of each transaction is scattered over the *Run-time Log* file, an information analysis approach is required to reduce the complexity of understanding the extracted information.

The first step of this analysis is to describe each extracted transaction based on its flow, data, and type within its lifetime. This requires to isolate for each single transaction its corresponding information from other transactions. As a transaction object is passed as a function argument to a communication interface (*b_transport* or *nb_transport*) by reference, this address can be used to trace the transaction in the *Run-time Log*. However, the reference address may be re-used for new transactions as soon as an old one is discarded (i.e. no two transactions can share the same address simultaneously). Thus, the type of modules and transaction's related parameters (for AT model) are used to detect its start and end points. By this, detailed information of each transaction's lifetime is stored in the *Trans lifetime* (Fig. 2, Phase 2).

Since it is possible that many of the extracted transactions in *Trans lifetime* have the same flow and type (only their data is different), a further analysis step is required to only visualize those which present a unique behavior. This effectively reduces the number of generated UML diagrams, allowing designers to quickly understand the behavior of a given TLM-2.0 design.

**Classifying Transactions**: The transactions' classification is performed in two levels: first based on the transactions' flow (providing designers with an abstract view) and then type (an accurate analysis). In order to classify the transactions based on their flows, the *Trans Lifetime* file is analyzed by the *Trans Classifier* module. This analysis is performed by generating a *Communication Pattern String* (CPS) for each transaction's lifetime stored in the *Trans lifetime*. For a given transaction, the CPS is a string of characters generated by concatenating the root and instance names of all modules taking part in the transaction's lifetime. For example, the CPS for the transactions generated by the *Initiator_A* (Fig. 1) to access *Target_A* through *AT_BUS* is "`Initiator_A:init_0+AT_BUS:bus_0-+Target_A:target_0`". By this, transactions in the *Trans lifetime* are categorized into several sub groups considered as *Unique Flow Group* (UFG) where each group presents a unique flow (communication pattern).

The next step of the transactions' classification is to perform a transaction type analysis in each UFG. For each transaction of a UFG, first the type of timing model is identified (LT or AT) based on the type of communication interface (*b_transport* or *nb_transport*). While the LT model can only be implemented in one way due to the TLM-2.0 base protocol, the AT model requires further analysis as it can be implemented in 13 unique ways. In case of the AT model, we take advantage of the transactions' related parameters to distinguish different types of the base protocol transactions in each group. To do this, a *Transaction Type String* (TTS) is generated for each AT model transaction's lifetime in a unique communication group. The TTS includes a concatenation of the communication interface(s), return status(es) and transition phase(s) of an AT model transaction. For example, the TTS of the $T_1$ transaction type (Table 1) generated by the *Initiator_A* module (Fig. 1) is "`fw/bw+TA/TU+BRQ/BRP/ERP`". Therefore, each UFG is divided into several sub groups where in each group the transactions have the same TTS. The UML diagram is generated for a transaction in each UFG that has a unique type. The final classified results are stored in the *classified Trans* by the *Trans Classifier* module (Fig. 2, Phase 2).

# 4 EXPERIMENTAL RESULTS

The *Static Info Analysis* module is implemented using the LibTooling library of Clang compiler [12]. The *Dynamic Info Analysis* and *Trans classifier* modules are implemented using C++ language. The proposed approach is applied to several standard VPs provided by Doulos [3] and [15]. All the experiments have been carried out on a PC equipped with 8 GB RAM and an Intel core i7 CPU running at 2.4 GHz. The experimental results of applying the proposed

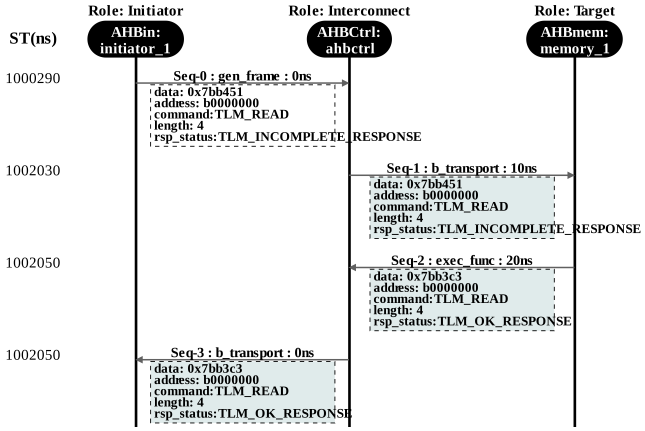


**Figure 4: UML diagram of SoCRocket-VP's LT transaction.**

approach to different types of SystemC VPs are shown in Table 2. The first two columns list names and lines of code for each VP, respectively. Column *TM* presents the timing model (LT or AT) of each VP. Column *#Trans* illustrates the number of each TLM-2.0 design's extracted transactions. Column *#UTrans* shows the number of unique transactions categorized by the number of unique flows (*F*), types (*T*) and generated UML diagrams (*UML*). Column *ET* list the execution time of the proposed approach followed by the required time for static (*P1*) and dynamic (*P2*) analysis. Column *CET* shows the pure compilation (*C*) and execution time (*E*) of each VP by GCC without any instrumentation. In the following we evaluate the quality of the proposed approach using a real-world case study.

**SoCRocket:** The proposed approach is utilized to extract both structure and behavior of the VP and generate UML sequence diagrams of the VP's unique transactions from the extracted run-time traces. As illustrated in Table 2, the proposed approach retrieved 7k transactions (5k LT and 2k AT model) from the VP. The results (column *#UTrans*) of the transactions' classification analysis (using *Trans Classifier* module) on the extracted transactions' lifetime (*Trans Lifetime* file) shows that, the VP includes 19 unique flows (communication patterns) and overall eight (one LT model and seven AT model) different types of base protocol transactions. Our method generated 21 UML diagrams for design understanding goal in less than a minute. Therefore, instead of reading the 50k lines of code of the VP distributed over more than 45 files, a simple glance over the quickly generated UML diagrams can significantly facilitate the design understanding and analysis process. The classified presentation of transactions' lifetime is stored in the *Classified Trans* to help designers for further possible analysis (e.g. debugging or validation) in the design process.

Fig. 4 shows the UML diagram of a single LT transaction's type of the SoCRocket VP in detail. The black shapes present root and instance name of modules within the design. The role (type) of each module is shown on top of the modules' name. The information on each arrow demonstrates interaction between two modules that is drawn from the caller to the callee w.r.t the simulation time. In particular, for a call from an instance of a TLM module, it presents the number of sequence, the name of the caller function, the timing phase (for AT model), timing annotation, and the return value of the callee (if available). Moreover, the generated UML model includes detailed transaction data. The box under each arrow shows the transaction's attributes which is passed as an argument from caller to callee. The white box illustrates a local transaction object while the blue boxes demonstrate a transaction object reached the callee through a function call. E.g. `seq-2` in Fig. 4 contains the

**Table 2: Experimental Results for all Virtual Prototypes**

| VP Name | LoC | TM | #Trans | #UTrans | | | ET (s) | | | CET (s) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | F | T | UML | P1 | P2 | Tot | C | E | Tot |
| LT-example[1] | 175 | LT | 500 | 1 | 1 | 1 | 1.4 | 0.53 | 1.93 | 1.3 | 0.1 | 1.4 |
| Example-4[1] | 547 | AT | 1000 | 2 | 4 | 5 | 2.5 | 0.2 | 2.7 | 1.8 | 0.1 | 1.9 |
| Example-5[1] | 650 | AT | 1000 | 8 | 2 | 9 | 3.2 | 0.3 | 3.5 | 2.1 | 0.2 | 2.2 |
| AT-example[1] | 2942 | AT | 1000 | 12 | 8 | 14 | 27.5 | 1.25 | 28.75 | 21 | 0.2 | 21.2 |
| Locking-two[1] | 3831 | LT/AT | 1000 | 14 | 10 | 16 | 29.2 | 1.85 | 31.05 | 24 | 0.3 | 24.3 |
| SoCRocket[2] | > 50000 | LT/AT | 7000 | 19 | 8 | 21 | 53.8 | 4.79 | 58.59 | 27.6 | 2.29 | 29.89 |

[1] Provided by [3] [2] Provided by [15] **LoC**: Lines of Code **TM**: Timing Model **LT**: Loosely-timed model **AT**: Approximately-timed model **#Trans**: number of Transaction **#UTrans**: number of Unique Transaction **ET**: Execution Time **CET**: Compilation and Execution Time by GCC without any instrumentation

information related to the response of the target module `AHBmem`. This information is the name of the called function (`exec_func`) and timing annotation (`20 ns`). It also includes the transaction data passed to the callee module (`AHBCtrl`) including the reference address of the transaction (`0x7bb3c3`), address (`b0000000`), command (`tlm::TLM_READ`), length (`4`), and response status (`tlm::TLM_OK_RESPONSE`).

## 5 CONCLUSION

The proposed approach provides designers with a fast solution to retrieve a significant amount of information describing the structure and behavior of a given VP. The approach is based on analyzing the AST of the design to extract static information and generate instrumented source code for run-time data extraction. The extracted information is translated into a structural and graphical representation, allowing designers to quickly understand the intricacies of VPs. We showed the effectiveness of the approach on several standard VPs including a real-world system. In future, we plan to extend the suggested information extraction and analysis approach for tasks of validation and verification at ESL.

***Acknowledgments:*** This work was supported by the German Federal Ministry of Education and Research (BMBF) within the projects SATiSFy under grant no. 16KIS0821K, SecRec under grant no. 16K1S0606K, CONVERS under grant no. 16ES0656, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

## REFERENCES

[1] 2006. IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, 1–423.
[2] John Aynsley (Ed.). 2009. *OSCI TLM-2.0 Language Reference Manual.* Open SystemC Initiative (OSCI).
[3] John Aynsley. Accessed: 2018-06-30. TLM-2.0 Base Protocol Checker. https://www.doulos.com/knowhow/systemc/tlm2.
[4] Harry Broeders and René Van Leuken. 2011. Extracting behavior and dynamically generated hierarchy from SystemC models. In *DAC*. 357–362.
[5] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. 2004. ParSyC: An Efficient SystemC Parser. In *SASIMI*. 148–154.
[6] Christian Genz and Rolf Drechsler. 2009. Overcoming limitations of the SystemC data introspection. In *DATE*. 590–593.
[7] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. 2016. AIBA: an Automated Intra-Cycle Behavioral Analysis for SystemC-based Design Exploration. In *ICCD*. 360–363.
[8] Mehran Goli, Jannis Stoppe, and Rolf Drechsler. accepted 2018. Automated Non-intrusive Analysis of Electronic System Level Designs. *TCAD* (accepted 2018).
[9] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. 2003. Efficient Automatic Visualization of SystemC Designs.. In *FDL*. 646–658.
[10] Anirudh Kaushik and Hiren D. Patel. 2013. SystemC-clang: An open-source framework for analyzing mixed-abstraction SystemC models. In *FDL*. 1–8.
[11] Wolfgang Klingauf and Manuel Geffken. 2006. Design structure analysis and transaction recording in SystemC designs: A minimal-intrusive approach. In *FDL*.
[12] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *BSD*. 1–2.
[13] Kevin Marquet and Matthieu Moy. 2010. PinaVM: a SystemC front-end based on an executable intermediate representation. In *EMSOFT*. 79–88.
[14] Tim Schmidt, Guantao Liu, and Rainer Dömer. 2016. Automatic Generation of Thread Communication Graphs from SystemC Source Code. In *SCOPES*. 108–115.
[15] Thomas Schuster, Rolf Meyer, Rainer Buchty, Luca Fossati, and Mladen Berekovic. 2014. SoCRocket - A virtual platform for the European Space Agency's SoC development. In *ReCoSoC*. 1–7, http://github.com/socrocket.
[16] Jannis Stoppe, Robert Wille, and Rolf Drechsler. 2013. Data extraction from SystemC designs using debug symbols and the SystemC API. In *ISVLSI*. 26–31.